2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

How Reliable Is My Wearable: A Fuzz Testing-based Study

Edgardo Barsallo Yi, Amiya K. Maji, Saurabh Bagchi
Purdue University
West Lafayette, IN, USA
{ebarsall, amaji, sbagchi}@purdue.edu

***Abstract*—As wearable devices like smartwatches and fitness monitors gain in popularity and are being touted for clinical purposes, it becomes important to evaluate the reliability of Android Wear OS and apps on such devices. To date there has been no study done by systematic error injection into the OS or the apps. We address this gap in this work. We develop and open source a fuzz testing tool for Android Wear apps and services, called Qui-Gon Jinn (QGJ). We perform an extensive fault injection study by mutating inter-process communication messages and UI events and direct about 1.5M such mutated events at 46 apps. These apps are divided into two categories: health/fitness and other. The results of our study show some patterns distinct from prior studies of Android. Over the years, input validation has improved and fewer `NullPointerExceptions` are seen, however, Android Wear apps crash from unhandled `IllegalStateExceptions` at a higher rate. There are occasional troubling cases of the entire device rebooting due to unprivileged mutated messages. Reassuringly the apps are quite robust to mutations of UI events with only 0.05% of them causing an app crash.**

I. Introduction

Over the last few years, wearable devices have grown tremendously in popularity. In the private consumer market, wearables include items such as smart glasses, smart watches, hearables, fitness and health trackers, smart jewelry, and smart clothing. The most successful wearable devices on the market today are smart watches and health and fitness trackers. The number of connected wearable devices worldwide was estimated to be 325 million at the end of 2016 and is expected to grow to over 830 million in 2020 [1]. Smart watches and health sensors today are providing a window into our health. However, every new technology brings with it new risks and reliability concerns and these have not been adequately studied for the wearables. In this paper, our objective is to explore the reliability of these highly personalized devices that are collecting deeply personal and in cases, sensitive, data. We also point to fundamental software engineering and architecture work that will make the apps more reliable.

While a few researchers have explored the design decisions and their vulnerabilities in the context of wearables [2], [3], they have not shed light on failure mechanisms and their propagation, in a detailed and comprehensive study. Compared to smart phones, wearable devices pose several new reliability challenges to device manufacturers and software developers. Limited display area, limited computing power, limited volatile and non-volatile memory, non-conventional shape of the devices, abundance of sensor data, complex communication patterns of the apps, and limited battery size—all these factors can contribute to salient software bugs and failure modes. Moreover, since many of the wearable devices are used for health purposes (either monitoring or treatment), their accuracy and robustness issues can give rise to safety concerns. Our paper, therefore, focuses on a systematic evaluation of popular apps on Android Wear (AW) devices, with a special focus on health/fitness apps.

There has been significant work on understanding the vulnerabilities of Android OS and its apps, adapting techniques from software testing [4]–[6]. However, to the best of our knowledge, we present the first study of the reliability of AW applications. AW shares much of the codebase with Android OS and follows Androids conventional programming paradigm: they are written in Java, compiled ahead-of-time, and executed atop the managed Android Runtime. However, there are major differences between traditional Android apps and Wear apps. This is primarily due to the smaller display area and rich sensor data in AW, as mentioned above. The richness of the user activities supported is much more limited and Wear apps mostly run in the background, communicate with users using notifications, and are typically controlled by a companion app on the smartphone. Based on these observations, we ask the following questions about reliability of AW:

1) *Exception Types*: What are the key differences between traditional Android apps and Wear apps in terms of exception handling? Are the relative proportions of manifestations of uncaught exceptions (such as, app crash or system reboot) similar?
2) *Failures across Applications*: What are the differences between failure manifestations across application types? Are health/fitness apps more or less robust than other apps?
3) *Robustness*: How well do Wear apps handle unexpected interactive user inputs? Can a user-level process crash the system?

To answer these questions, we present the design and implementation of **Qui-Gon Jinn (QGJ)**[1], a simple user-level tool (*i.e.,* does not need rooted devices) for testing robustness of Android Wear apps. QGJ consists of two components—a) QGJ-Master[2]: An Android app for sending generated inter-

[1]Named after a powerful but maverick Jedi Master from Star Wars.
[2]Unless otherwise specified, we use the names QGJ-Master and QGJ interchangeably. The UI testing component is explicitly referred to as QGJ-UI.

2158-3927/18/$31.00 ©2018 IEEE
DOI 10.1109/DSN.2018.00050

component communication messages (intents) to targeted applications, and b) QGJ-UI: A tool based on Android Monkey fuzzer [7], which can send mutated (malformed) UI events to the wearable device. Using both these components, we systematically inject a large number of mutated, synthetic intents or mutated UI events to a selection of 45 popular Wear apps. We then analyze the system logs to find how well these applications handled the mutated intents or events.

Our key findings from the study are:

1. Distribution of exception types does indeed differ between Android [8] and Wear apps. Over the years, input validation in Android has improved and fewer `NullPointerException`s are seen, however, AW apps suffer failures at a higher rate from unhandled `IllegalStateException`.
2. Across application types, built-in apps showed more failures compared to third-party apps. This has worrying implication for error propagation since many Android apps reuse built-in apps and components. Our results did not indicate any significant difference between health/fitness apps and other apps.
3. Several times during the experiments, the Wear device rebooted in response to malformed intents. These reboots did not occur in response to a single "deadly" intent but rather at specific states of the device due to escalation of multiple errors. This would indicate that the malformed intents caused error accumulation, which eventually rebooted the system.
4. AW apps offer significant scope for improvement of input validation. Although, many input validations are performed by the system (which is an encouraging sign), apps are not taking full advantage of these routines. Automated robustness testing tools (such as QGJ) can help in detecting such bugs and bridging this gap.

The rest of the paper is organized as follows. First, we present an overview of Android in Section II. Then in Section III, we present the design QGJ and our experimental setup. Section IV discusses the results followed by related work in Section V. Finally, we conclude the paper in Section VI with a discussion of threats to validity.

II. Background

A. Intents and Application Components

The Android programming model is based on passing intent messages for communication within or across applications. An intent can been seen as a passive data structure with an abstract description of an operation to be performed. The basic information in a intent includes: *Action*: a defined action specified in the Android API, such as `ACTION_VIEW`, `ACTION_EDIT`, `ACTION_DIAL`, etc.; *Data*: an URI that represents the data item to be operated on, such as a website URL, or the URI of a contact. Additional optional fields includes: *Category*: additional information about the action to execute; *Type*: specifies the explicit type for intent data (a MIME type); *Component*: indicates the component class to be used for the intent; *Extras*: key value pairs for additional information. Moreover, there are two types of intent: *implicit* and *explicit*, depending if the target or destination component is defined in the intent. Implicit intents are delivered to the best matching component in the system, based on the information contained in the intent. QGJ is focused on explicit intent.

The two types of Android application components relevant to this study are as follows. *Activity*: the entry point for interacting with the user. One app can invoke an activity in another app, if permissions are granted. *Service*: a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

B. Android Wear

Android Wear (AW) is the version of Android OS designed for smartwatches and other wearable devices. The first version was released in 2014, and AW 2.0 in early 2017. Unlike Android, AW is not completely open source; but is one of the most popular OSes for wearables, with Apple WatchOS and Samsung Tizen. Contrary to mobile phones, wearables require minimal human interaction (micro transactions). Hence, AW user interface (UI) is designed to be the least attention seeking to the user, by showing minimal information and centered on notifications, watch faces, native applications and voice commands. AW applications are more services driven in contrast to Android applications, which usually have rich GUI. Besides, since wearables need to be worn on the body, AW take advantage of context-awareness by sensing information from hardware and software sensors equipped on the devices.

III. Design of our Tool and Experiment

Our testing tool, Qui-Gon Jinn (QGJ), consists of two distinct components: a) QGJ-Master, a generational fuzzer that sends IPC messages to various apps, and b) QGJ-UI, a mutational fuzzer that sends mutated UI events to the wearable. Among these two, QGJ-Master is more complicated and its design is presented in sections III.A-III.D. The design of QGJ-UI and its experimental setup is presented in Section III.E. In the following, we use the terms QGJ and QGJ-Master interchangeably.

A. Testing Tool

QGJ follows the architecture of Jar Jar Binks (JJB) [8], an Android robustness testing tool, which exploits IPC on Android phones. The tool can fuzz a single component or a group of components registered in a device. JJB supports fuzz injection of Activities, Services, and Broadcast Receiver components. QGJ extends the JJB capabilities to support Android Wear, thus broadening the applicability significantly to wearables. We have open sourced our tool and hope that others will use it for benchmarking the emerging class of wearable apps [9]. The overall architecture of QGJ is shown in Figure 1. QGJ allows us to inject randomly generated intents to both mobile and wearable devices. Basically, the tool, which needs to be installed on two paired devices (mobile phone and wearable) consists of three main components: a mobile application (QGJ Mobile), a wear application (QGJ Wear), and a fuzzer library. We begin describing each component briefly.

In this particular evaluation, we are only focusing on fuzz testing on the wearable device. Henceforth, we may shorten this to simply saying "fuzz test".

QGJ Mobile. This is an Android application, which runs on the mobile and offers a UI to interact with the fuzzer. The user can choose the component type to fuzz (*Activities* or *Services*) and the type of test to execute (a specific campaign as outlined in Table I). The application allows the user to choose the target device: *mobile* or *wearable*. If the mobile device is chosen as target device, the fuzz test is done locally. Otherwise, QGJ Mobile communicates with the wearable device to orchestrate the fuzz test. Once the test is completed, the app shows a summary of the results.

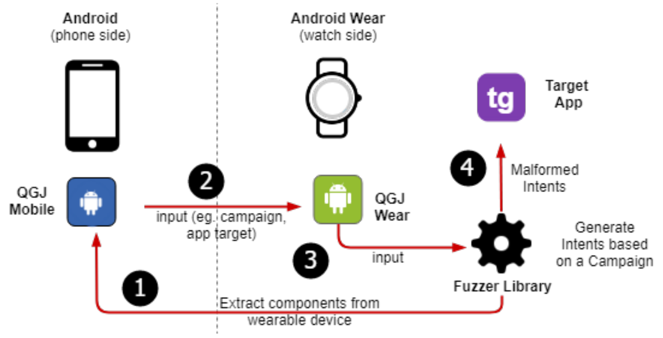
QGJ Wear. This is an Android Wear application, which executes on the wearable. It communicates with the mobile app using the Android Wear *MessageAPI* and *DataAPI*. The application receives the selected options on the UI, and executes the fuzz test using the `Fuzzer` library. After the fuzzing, the wearable app sends a summary of the results to the mobile application.

Fuzzer Library. This is the Java library, which contains the main functions needed to inject intents on the target device. Since intents have to be sent from the target device, this library is shared by *QGJ Mobile* and *QGJ wearable*. The library runs the fuzz experiments asynchronously in the target device, and the output is stored in the execution logs on the device.

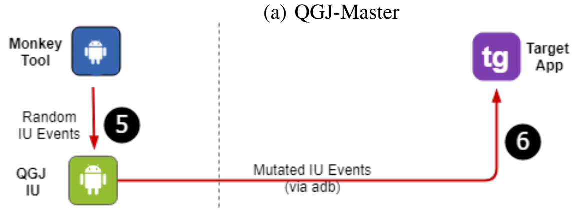
The communication between the mobile device and the wearable is shown in Figure 1a. The QGJ Mobile app first retrieves list of components (*Activities*, *Services*) from the Android wearable (❶). Next, from the Android device, we choose a target application and the fuzzing campaign to use. Then the Android phone communicates with the wearable using the AW MessageAPI (❷). When the AW device receives the message, the wearable app forwards the input (the target component and the fuzz campaign) to the Fuzzer library to initiate the intent injection, ❸. The fuzzer library, which is part of the QGJ wear application, triggers the fuzzing on the chosen target app component (❹). One of the goals of QGJ is to keep the tool simple and broadly usable. Therefore, QGJ does not need any root privilege on the device to run.

B. *Fuzz Intent Campaigns*

Table I summarizes the Fuzz Intent Campaigns (FIC) used for our evaluation. The table includes the strategy used to generate intents, the number of intents generated, and a sample injected intent for each fuzz campaign. The fuzzer has over 100 different Actions and 12 types of data URI (*e.g.,* https, http, tel) configured. Combinations of these are used in the intents generated during various FICs. We use these FICs to evaluate how different kinds of corruption, from the subtle to the more egregious, affect how they are handled by the app components. In the best case, these should be handled "gracefully" and not cause a user-visible failure. FICs are targeted to *Activities* and *Services* components because they form the large majority of the components on AW apps. It is therefore important to understand how resilient they are to incorrect intents.


Android
(phone side)
Android Wear
(watch side)
Target App
QGJ Mobile
input (eg. campaign, app target)
QGJ Wear
Malformed Intents
Generate Intents based on a Campaign
input
Fuzzer Library
Extract components from wearable device

(a) QGJ-Master



(b) QGJ-UI

Fig. 1. Operational workflow of using QGJ for doing fuzz-based injection of apps on the mobile device (a phone is shown here) or the wearable (a watch is shown here). The workflow shows the communication between the phone and the watch. The diagram does not include a target app on the mobile side for clarity since this study is focused on fuzzing Android Wear apps.

C. *Target Applications and Error Manifestations*

Target apps. As described in Section II-B, we define two categories of applications—*Health/Fitness* and *Not Health/Fitness*. This categorization is suggested by the fact that health/fitness apps are unique to the AW ecosystem. These are aimed to monitor user activity through the use of hardware or software sensors included in the wearable device. In most cases, these apps interact with the Google Fit API to access the sensors. This dependency could mean that *Health/Fitness* apps are susceptible to propagation errors from the Google Fit API, a hypothesis that we verify through our experiments.

Applications can be orthogonally classified as either built-in or third party apps. This distinction is important to note because built-in apps were developed by Google and are already pre-installed on the device. These apps could include general purpose applications (*e.g.,* Google Calendar, Gmail) or Android Wear core apps, like Google Fitness. Third-party applications can be downloaded from the Google Play Store and installed by the user. We used the number of downloads for third-party apps as a measure of the maturity and popularity levels of applications and selected only those with greater than 1 million downloads. The complete breakdown of the apps selected for our study is shown in Table II. The table includes the number of components (Activities and Services) per category. Previous to the start of the experiments, we chose the 35 third-party apps from the Google Play store and installed them in the smartwatch.

Error manifestations. For the experiments, we defined four possible reliability manifestation or behaviors, which we list in decreasing order of severity.

TABLE I
FUZZ INTENT CAMPAIGNS

Campaign	Characteristics of Intents Generated	# Intents Generated	Intent Example
A: Semi-valid Action and Data	Valid `Action` and valid `Data` URI are generated separately, but the combination of them may be invalid.	|`Action`| × |`TypeOf(Data)`| 1M aprox. Intents overall	`{act=ACTION_DIAL, data=http://foo.com/, cmp=some.component.name}`
B: Blank Action or Data	Either `Action` OR `Data` URI is specified for each Intent, but not both. All other fields are left blank.	|`Action`| + |`TypeOf(Data)`| 100K aprox. Intents overall	`{data=tel:123, cmp=some.component.name}`
C: Random Action or Data	`Action` or the `Data` URI is valid, and the other is set randomly.	|`Action`| + |`TypeOf(Data)`| 300K aprox. Intents overall	`{act=ACTION_DIAL, cmp=some.component.name}`
D: Random Extras	For each `Action` defined, we create a valid pair `{Action, Data}` with a set of 1-5 `Extra` fields with random values.	|`Action`| 250K aprox. Intents overall	`{act=ACTION_DIAL, data=tel:123, cmp=some.component.name (has extras)}`

TABLE II
APPLICATION STATS

Category	Classification	#	# Activities	# Services
Health/Fitness	Built-in	2	81	34
Health/Fitness	Third Party	11	80	59
Not Health/Fitness	Built-in	9	168	188
Not Health/Fitness	Third Party	24	185	117
Total		**46**	**514**	**398**

System reboot. The Operating System reaches an unrecoverable state and the device reboots. The reboot can also be confirmed in the log files collected from the target device. This is a serious manifestation because it can be used to launch a Denial-of-Service against the entire device.
Crash. The application crashes due its inability to handle malformed intents. This behavior is identified in the log files as a "`FATAL EXCEPTION: main`" entry.
Hang or unresponsive. The application experiences temporary unresponsiveness or freezes permanently, and does not respond to any action. Eventually a correct state is achieved, in some cases with human intervention. This manifestation is distinguish by an `ANR` (Application Not Responding) error in the log files.
No effect. There is no effect or failure manifestation due to the malformed intent. The application and the OS behave as expected. We can verify this behavior from the log files by the absence of exceptions or errors, or a `SecurityException` triggered by the OS after receiving the malformed intent.

D. Experiment Setup

The FIC experiments were conducted using a phone (LG Nexus 4) paired via bluetooth to a wearable device (Moto 360, running Android Wear 2.0 released in Feb 2017). QGJ was installed on both devices. The phone was used to choose the required input and start the FICs intended to fuzz the applications installed on the smartwatch. No injections were performed on the phone. Prior to the experiments, we tested all the applications on the wearable device to check for basic device-app compatibility. Moreover, we performed any initial setup required by the apps, to ensure that all the functionalities are available during the experiments. One important point of departure in our app components is the relatively high frequency of Services compared to Activities. Previous studies had targeted Activities at a higher rate since they are more numerous in regular Android applications. Since the user interaction with wearable apps is usually shorter as compared to mobile apps, most of each application's workload is done by Services, which are background running components that do the work triggered by some user action.

During the experiments, over a million and half intents were sent to over 900 components (between Activities and Services). The mechanism to run the experiments is as follows. First, we choose a particular wearable application using QGJ UI, from the mobile phone, and begin the experiments. The fuzzer starts injecting malformed intents according to the particular FIC. All 4 campaigns are executed one after another. Once the execution of the experiments is done, we collected all of the log files (over 2GB) from the wearable using `logcat`, through the `adb` interface. Then, we analyzed the logs to gather information, and for each component classified the behavior of the application according to the expected scenarios described in Section III-C. For any failure or error encountered, we manually analyzed further the logs files to find their possible root cause. The QGJ fuzzing model is based on injection of random intents into the wearable app, following a pattern defined for each experiment. To keep the load due to intents realistic, we insert two delays: (a) 100 ms between successive intents similar to JJB; and (b) 250 ms after every 100 intents. It was empirically determined by checking the logs that these delays were required to ensure the device is not overloaded.

Since previous works targeted earlier version of Android, we decided to run similar experiments on a mobile phone to have a more accurate comparison between the Android and AW ecosystem. The experiments included all four campaigns, targeting a Nexus 6 running Android 7.1.1. We focus our attention on common applications pre-loaded on the phone (e.g. Google Chrome, Google Play Store), which are often use by third party application for implementing common functionalities. After filtering the apps by the prefix `com.android`, we found 63 apps (595 Activities and 218 Services).

E. QGJ-UI Design

While QGJ-Master evaluates the robustness of AW applications by sending explicit intent messages, it does not test how apps handle user interactions. In most cases, sending an intent via QGJ-Master has the effect of launching an activity or a service. However, after launching an application, users often interact with the application (or the device underneath) either via touchscreen or via hardware keys. Evaluating application robustness against such interactions would require emulating user interface (UI) events. For this purpose, we developed

QGJ-UI based on the Android stress testing tool Monkey [7], which mutates intents or user events resulting from UI actions.

Figure 1b depicts QGJ-UI workflow. First, monkey is run on the target device to generate a specific number of UI events (❺). For event generation, we specify equal percentages for different types of events (e.g. touch, trackball, app switch, permission etc.). These UI events may trigger monkey to generate some intents. Next, the monkey logs are parsed to find the UI events and intents sent to the wearable (❻). Similar to QGJ-Master, QGJ-UI generates two types of mutated events – semi-valid and random. In semi-valid, the arguments for an event are randomly replaced by another valid value for that argument that had been observed during the experiment. Similarly, for random events, the arguments are replaced with a random ASCII string or a float value (depending on type). An example random event would be: `input tap -8803.85 4668.17` (note the invalid X, Y coordinates). These mutated events or mutated intents are then sent to the target device using `adb shell` utilities.

For this experiment, we used an Android Watch emulator (Android 7.1.1, API level 25) and paired it with a Nexus 6 phone (Android 7.1.1). The choice of the Watch emulator instead of the actual watch from the QGJ-Master experiment was so that we could study the core functionality in isolation (which is incorporated into the emulator) rather than together with the vendor-specific extensions (as would be present in the actual watch). Moreover, since different watches have varying screen sizes and shapes, the same UI event may execute widely different actions across devices. Using an emulator helps us avoid that and run repeatable experiments. Similar to the apps listed in Table II, we installed on the emulator all the built-in apps and the top 20 of the most popular third-party apps. Although we do not target the phone for this study, several Wear apps caused UI components to pop-up on the phone. Logs from the emulator were collected using `logcat` and later analyzed to generate the results, which we present in Section IV-D.

IV. Experimental Results

We discuss our results based on the following perspectives: (i) distribution of the error manifestations in the apps in response to fuzzed intents; (ii) distribution of exceptions and their ultimate error manifestations.

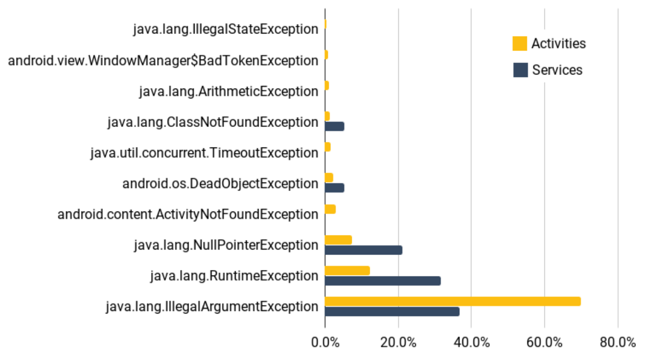


Fig. 2. Distribution of type for uncaught exceptions (without considering Security Exception) grouped by component type.

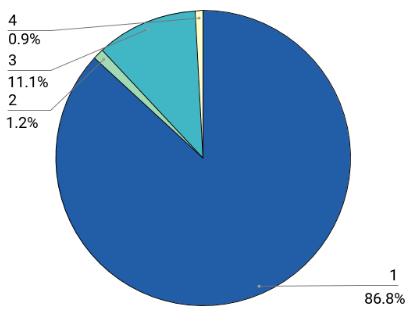
A. Distribution of Exception Types

To understand how well Android Wear responds to malformed intents, we measure the distribution of uncaught exceptions over all FICs. Here each exception is counted once per component, even if it was raised several times. Fig. 2 shows the distribution without considering security exceptions, which represent 81.3% of all exceptions. Some intents are reserved for privileged OS processes and when sent by QGJ raises the security exception. For example, when QGJ sends an intent {`act=ACTION_BATTERY_LOW`}, a `SecurityException` is thrown and the intent is ignored by AW. This is the specified and secure behavior. After `SecurityException`, the second largest share belongs to `IllegalArgumentException`. This type of exception is raised because of the mismatch on the data contained in an injected intent and what is expected by the component.

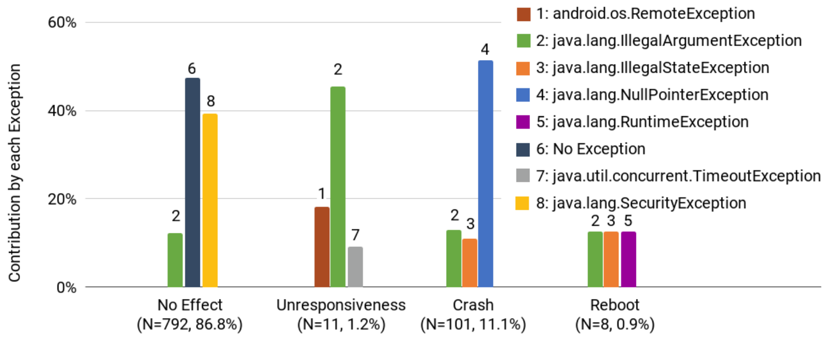
We next consider the proportion of application components that are affected by the mutated intents and classify the components according to the 4 manifestations. This is shown in Figure 3a. If a component has different manifestations to multiple injected intents, we take the most severe manifestation. We find that almost 90% of the components are not affected at all. The most dominant error class is crash, which is more than 8X the next error class, unresponsive. The most severe error class, device reboot, affects 4 of the components.

Next, for each error manifestation, we study what exceptions are the ultimate cause of that manifestation. This is shown in Figure 3. The exception to which an error is ascribed is the one that we determine through a simplified and semi-automatic root cause analysis. With many cases a simple temporal chain is used to determine the root cause automatically—thus the first exception in a chain of exceptions is assigned the guilt (e.g. in the case of `RuntimeExceptions`). In some cases, a tight-knit pattern among the exceptions is deduced and one cannot be inferred to causally precede the others. In such cases, we assign the blame for that error manifestation equally among the exception classes. The first observation we make is that the `NullPointerException` still dominates the crash cases, as in all prior studies on Android reliability [5], [8], [10]. However, the relative proportion is less and the decrease has been taken up by an increase in the proportion of `IllegalArgumentException` and `IllegalStateException`. For the no effect case, in about 90% of the cases, there is no exception thrown upon receipt of the injected intent. In the remaining 10% of the cases, an exception is thrown but that is handled by the app gracefully. For the pathological case of device reboot, three exception classes are equally culpable. For the unresponsiveness error category, `IllegalStateException` dominates, while the presence of `android.os.DeadObjectException` hints that garbage collection can have the undesirable effect.

Furthermore, a crash due to `ArithmeticException` is worth highlighting. First, the `ArithmeticException` was reported by a *Health & Fitness* application because a ”divide by zero” operation was reported on an AW class



(a) Distribution of error manifestations



(b) Distribution of exceptions by manifestation

Fig. 3. (a) Distribution of error manifestation among all the app components targeted by QGJ. The legend corresponds to each of the states: (1) No Effect, (2) Unresponsive, (3) Crash, (4) Reboot. (b) Distribution of the most frequent exceptions for each error manifestation. The percentage of each exception is with respect to the total exceptions per manifestation. The number at the base indicates the absolute number of components showing such manifestation and the percentage is with respect to the total number of injected components.

`GridViewPager`. This Layout Manager class, which allows navigation in both axes, was deprecated in AW 2.0 in favor of other classes since horizontal paging is not encouraged anymore [11], [12]. This finding indicates the presence of errors in Android Wear ecosystem due to the lack of migration to the AW 2.0 specification of some applications.

Next, we turn our attention to the numerous `IllegalArgumentExceptions`. This exception should be thrown to indicate that an argument is either illegal or inappropriate. It is not surprising to find this exception in the logs; however, this exception should not cause the application to crash. A crashing behavior would indicate that input validation in the activity has been implemented only partially. For instance, Google Fit, a core AW component, reported a crash because an intent {`act=ACTION_ALL_APP`} was sent without the expected message (*Complication Provider*).

B. Distribution of Error Manifestations

Table III presents the distribution of behaviors for all applications over the four fuzzing campaigns, grouped by application type (as *Health/Fitness* or *Not Health/Fitness*). Here we classify the effect of the injection on an entire application according to the four error manifestations. Since different components within an app can have different manifestations, we use the most severe manifestation for this result. We conclude that there is *no* clear indication that *Health/Fitness apps*, due to implicit complexity because of dependence on other components (*e.g.,* Google Fit API), are less robust than others apps. Both categories have no effect due to the injection at roughly the same rate, 69.2% for health apps versus 74.5% for others, and both reported device reboots.

Fig. 4 presents the exceptions that cause the crashes, broken down by built-in and third-party apps. The percentage is calculated taking the two application classes together. It is noteworthy that built-in apps reported crashes at a higher rate (64%) than third-party apps (46%). The failures included those in built-in core AW components aimed to track workout activity, such as Google Fit and Motorola Body.

During the fuzzing campaigns, the system restarted twice due to crashes. We find empirically that this manifestation depends on the transient state of device and happens with error propagation across components and due to software aging through repeated fuzzing campaigns. We give the detailed post-mortem of these two cases considering the severity of this error manifestation. First, a sequence of malformed intents to a health app, which interacts with heart rate sensor using `SensorManager` class (rather than the more common Google Fit) provoked a system restart. There were no exceptions raised before the crash, which means the malformed intents were not rejected by the app. During the sequence of injections, the application experienced unresponsiveness (`ANR`) which explains the `SIGABRT` sent by the system to shutdown the SensorService process `/system/lib/libsensorservice.so`. Since this is the core process which handles Sensor access on AW, the system was left in an unstable state and the device rebooted. The second device reboot was due to the inability of the system to start an Activity because of missing data in the malformed intent injected in a built-in app. The application crashed several times due to the inability to start the activity that prevented it from binding to the Ambient Service, a core AW service to control low-power ambient mode. Then, the system sent a `SIGSEGV`, which caused segmentation fault of the system process, that eventually ended up rebooting the device.

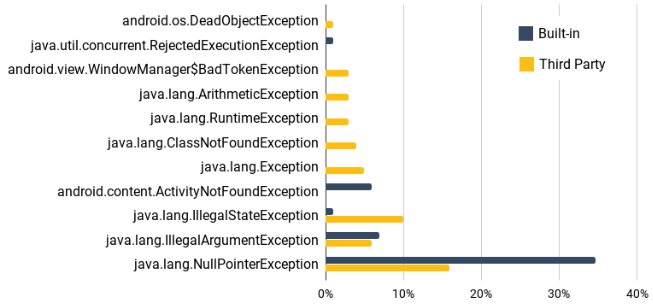


Fig. 4. Distribution of exceptions that originated crashes grouped by app classification.

C. Distribution of Crashes on Android Phone

In Table IV, we show the results after targeting the mobile phone (running Android 7.1.1) with the four FICs in QGJ. Similar to previous studies [5], [8], [10], the pri-

TABLE III
DISTRIBUTION OF BEHAVIORS AMONG FUZZ INTENT CAMPAIGNS

	Reboot		Crash		Hang		No Effect	
	Health	Not Health	Health	Not Health	Health	Not Health	Health	Not Health
A: Semi-valid Action and Data	8%	0%	23%	30%	8%	0%	62%	70%
B: Blank Action or Data	0%	0%	31%	24%	0%	0%	69%	76%
C: Random Action or Data	0%	0%	31%	33%	8%	0%	62%	67%
D: Random Extra	0%	3%	15%	30%	8%	0%	77%	67%

TABLE IV
DISTRIBUTION OF CRASHES ON ANDROID PHONE PER EXCEPTION TYPE. "OTHERS" CONTAIN EXCEPTIONS THAT CAUSED FEWER THAN 5 CRASHES.

Exception	#Crashes	%
java.lang.NullPointerException	54	30.9%
java.lang.ClassNotFoundException	46	26.3%
java.lang.IllegalArgumentException	31	17.7%
java.lang.IllegalStateException	10	5.7%
java.lang.RuntimeException	9	5.1%
android.content.ActivityNotFoundException	7	4.0%
java.lang.UnsupportedOperationException	6	3.4%
Others	12	6.9%

mary cause of crashes was `NullPointerException`. `ClassNotFoundException` had the second largest share, followed by `IllegalArgumentException` and `IllegalStateException`. The later two are also major reason of crashes in AW, as shown in Section IV-A. In contrast, `ClassNotFoundException` and `IllegalArgumentException` were more frequent on the phone. These results indicate that input validation on Android has improved over the years, and fewer uncaught `NullPointerException` are raised in Android 7.1.1 compared to results from Maji *et al.* [8].

D. QGJ-UI Results

In Table V we give the results of our injection into the UI events and resultant intents through the QGJ-UI, which was introduced in Section III-E. We only see two of the four categories introduced earlier, "Crash" and "No effect". Only 0.05% of the injections lead to app crash with semi-valid injections, while there is no crash with random injections. This is despite the fact that 1.5% of the injections lead to exceptions, but all of these are handled. Reassuringly, we did not observe any system crash during our UI injections. We found that compared to QGJ-Master, QGJ-UI showed much fewer number of exceptions and crashes, thereby showing better resilience to malformed UI events. We posit that, besides better input validation at the event handlers, two other factors contribute to these positive results—(i) QGJ-UI only sends intents to launcher activities of various applications, therefore, the set of target components is fewer in QGJ-UI. These components are also simpler and therefore tend to be more reliable; (ii) We found that various `adb` tools (subcommands) such as `shell input`, `am` (ActivityManager shell utility), `pm` (PackageManager shell utility) have robust input validation and sanitization routines. For example, if an activity `com.android.phone` is invoked by QGJ-UI using `am` without specifying an action or a category (similar to FIC B), `am` automatically sets the action and category values as {`act=action.MAIN cat=category.LAUNCHER`}. It is

TABLE V
DISTRIBUTION OF EXCEPTIONS AND CRASHES DURING QGJ-UI EXPERIMENTS.

Experiment	#Injected Events	Exceptions Raised	Crashes
Semi-valid	41405	1496 (3.6%)	22 (0.05%)
Random	41405	615 (1.5%)	0 (0%)

encouraging to see that UI event handlers and `adb` tools have better input validation and exception handling capabilities compared to other application components.

During our experiments, we also found that different Android components handle invalid inputs differently. For example, if the `pm` utility is asked to send a random permission string `'S0me.r@ndom.$trinG'` to `some.component`, it rejects the input string saying that no such permission exists. However, the `am` utility would forward the string `'S0me.r@ndom.$trinG'` as an action string to a component and relies on the correctness of input validation at the component. Although this did not lead to crashes, we suggest that input validation be more stringent, e.g., only permit action or category strings that are registered on the device.

E. Software engineering techniques for improving robustness

Based on the results, it can be seen that Wear applications often crash in response to unexpected intents, which cause uncaught exceptions. Although these results are better compared to [8] where `NullPointerExceptions` contributed to 46% of all exceptions, there is still significant scope for improvement. For example, `IllegalArgumentExceptions` are often not handled correctly by apps leading to crashes. From our insights, we suggest three software engineering techniques for improving robustness of Android Wear apps.

Better tool support: In the software engineering community, there has been significant amount of work to analyze exception handing in Java applications [13]. The high-level idea is to perform static analysis of the application codes to check how exception handling codes are linked together. Features like exception handling warning, the Analyze Stacktrace tool, and the Lint static code inspection tool in Android Studio IDE are steps in the right direction. Integration of Android Studio with dynamic testing tools like QGJ can further help developers to improve application robustness.

Research on software aging: During our experiments, we found the Adroid Watch rebooted twice. These reboots were not due to a single malformed intent, but rather manifested at certain stages of the experiments. We hypothesize that such reboot is a manifestation of error accumulation in the Android watch. We believe that research on software aging and rejuvenation can help detect and potentially recover from such accumulated errors. A recent work by Cotroneo *et* *al.* [14] supports our observation and suggests some metrics for detetcting software aging in Android.

V. Related Work

Android. Over the years, there have been several projects on testing of Android applications. At a high level, these can be broadly classified into papers that focus on Android security and those that focus on application robustness or correctness. In Android security testing, popular themes are—Android permission model [15], Inter-process communication vulnerabilities [16], and privilege escalation [17]. Among these, research related to inter-process communication in Android is closest to our work. Although we do not exploit IPC vulnerabilities, our testing attempts to find input validation errors in intra and inter application messages which may lead to detection of security vulnerabilities. Non security-related research can be broadly classified into two categories: (i) solutions that focus on testing application GUI in Android [6], [18], [19], and (ii) solutions that focus on finding bugs or design flaws in application components [4], [8], [20].

Our work is closest to the JarJarBinks (JJB) tool presented by Maji *et al.* [8]. JJB uses different types of fuzz injection campaigns (FIC) to detect exception handling errors in stock Android applications and services. Our paper differs from [8] in two significant aspects: (i) we focus on Android Wear apps instead of Android mobile apps and (ii) we not only inject intent messages, but also inject UI events.

Android Wear. Existing research work on AW primarily focuses on the efficiency, performance, and correctness of Wear runtime and applications. Liu *et al.* examined the execution of AW apps by profiling the OS and discovered execution inefficiencies and OS design flaws [3], [21]. Chen *et al.* [22] characterized various properties of smartwatch usage in the wild. Their study provides a better understanding of wearables, and characterizes key system aspects, such as power modeling and network behaviors. More recently, Zhang *et al.* presented a testing tool based on modeling the AW notification mechanism, geared to achieve high coverage [19]. However, to the best of our knowledge, our paper presents the first robustness study of AW applications. We found that in contrast to [8], Android Wear shows fewer crashes from `NullPointerExceptions` and more crashes from `IllegalStateExceptions`.

VI. Conclusion

Here we have done an extensive study of the reliability of Android Wear apps through mutating inter-process communication messages (called intents) and UI events. We find that while `NullPointerException` handling has improved over the years on Android, there is still a disturbingly high incidence of other exceptions, such as `IllegalStateException`. We find that built-in apps crash at a higher rate than popular third-party apps and a confluence of factors—software aging and cascading failures—can cause the entire device to reboot even through mutating unprivileged intents. The apps are remarkably resilient to mutating UI events with only 0.05% of them causing an app crash. We shed light on three approaches that can improve the resilience of AW apps—better IDE support, stronger input validation, and guarding against software aging.

We acknowledge three primary threats to validity. First, our study has used a single wearable device and thus is blind to vendor-specific customizations. Second, while most AW apps are two-part, with a mobile device and a wearable component, we have ignored the inter-device interactions and focused only on the wearable components. Third, our comparison with Android error manifestations does not include third-party apps. Our future work will focus on addressing these concerns.

References

[1] Statista, "Statistics & facts on wearable technology," 2017. [Online]. Available: https://www.statista.com/topics/1556/wearable-technology/

[2] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale, "Weardrive: Fast and energy-efficient storage for wearables." in *USENIX Annual Technical Conference*, 2015, pp. 613–625.

[3] R. Liu and F. X. Lin, "Understanding the characteristics of android wear os," in *Mobisys*, 2016, pp. 151–164.

[4] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *ISSRE*, 2017.

[5] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *ASE*, 2015, pp. 429–440.

[6] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *FSE*, 2013, pp. 224–234.

[7] Android. UI/Application Exerciser Monkey. [Online]. Available: https://developer.android.com/studio/test/monkey.html

[8] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in android," in *DSN*, 2012, pp. 1–12.

[9] "Qui-Gon Jinn: An Android Wear Benchmarking Tool." [Online]. Available: https://github.com/ebarsallo/QGJ

[10] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *WODA and PERTEA*, 2014, pp. 1–5.

[11] Google. (2017) Android developers. gridviewpager. [Online]. Available: https://developer.android.com/reference/android/support/wearable/view/GridViewPager.html

[12] Google. Android developers: Creating custom uis for wear devices. [Online]. Available: https://developer.android.com/training/wearables/ui/index.html

[13] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in java server applications," in *ICSE*, 2007, pp. 230–239.

[14] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono, "Software Aging Analysis of the Android Mobile OS," in *ISSRE*, 2016, pp. 478–489.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM CCS*, 2011, pp. 627–638.

[16] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Mobisys*. ACM, 2011, pp. 239–252.

[17] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, vol. 31, 2011.

[18] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Workshop on Automation of Software Test*, 2011, pp. 77–83.

[19] H. Zhang and A. Rountev, "Analysis and testing of notifications in android wear applications," in *ICSE*, 2017, pp. 347–357.

[20] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *ACM CCS*, 2015, pp. 1236–1247.

[21] R. Liu, L. Jiang, N. Jiang, and F. X. Lin, "Anatomizing system activities on interactive wearable devices," in *APSys*, 2015, pp. 1–7.

[22] X. Liu, T. Chen, F. Qian, Z. Guo, F. X. Lin, X. Wang, and K. Chen, "Characterizing smartwatch usage in the wild," in *Mobisys*, 2017, pp. 385–398.